Automating Object Transformations for Dynamic Software Updating via Online Execution Synthesis

Anonymous Authors

Abstract
Dynamic software updating (DSU) is a technique to upgrade a running software system on the fly without stopping the system. During updating, the runtime state of the modified components of the system needs to be properly transformed into a new state, so that the modified components can still correctly interact with the rest of the system. However, the transformation is non-trivial to realize due to the gap between the low-level implementations of two versions of a program. This paper presents AOTES, a novel approach to automating object transformations for dynamic updating of Java programs. AOTES bridges the gap by abstracting the old state of an object to a history of method invocations, and re-invoking the new version of all methods in the history to get the desired new state. AOTES requires no instrumentation to record any data and thus has no overhead during normal execution. We propose and implement a novel technique that can synthesize an equivalent history of method invocations based on the current object state only. We evaluated AOTES on software updates taken from Apache Commons Collections, Tomcat, FTP Server and SSHD Server. Experimental results show that AOTES successfully handled 51 of 61 object transformations of 21 updated classes, while two state-of-the-art approaches only handled 11 and 6 of 61, respectively.

Keywords and phrases Dynamic Software Update – Program Synthesis – Execution Synthesis

Digital Object Identifier 10.4230/LIPIcs...

1 Introduction
Today’s industry definitely requires high availability of software systems. One of the major losses of availability is caused by system shutdowns for installing software updates that fix bugs and security vulnerabilities. Dynamic Software Updating (DSU) can eliminate this loss by updating running software systems without stopping them.

Modern operating systems and programming language virtual machines provide powerful runtime code manipulation facilities such as dynamic linking [8], dynamic class loading [28], on stack replacement [36] and live patch [2]. With these facilities, it is not difficult to update the code of a running program. In addition to code replacement, DSU also needs to ensure that the new loaded code can execute properly with the existing runtime state in the memory (e.g., heap objects) after the dynamic update.

Existing DSU supporting systems, e.g., Gingseng [32], Jvolve [40] and Javelus [15], ensure only syntactical correctness, i.e., no type error would be caused by the update. To preserve semantics correctness, the DSU system should apply an additional state transformation that maps the runtime state left by the old code to a proper state with which the new code continues. To realize the state transformation, developers usually specify a manually-prepared state transformation function, i.e., state transformer.

However, it is difficult and error-prone to develop and test state transformers. Software is seldom developed with DSU in mind but is assumed to start from scratch. The internal states between different versions of a program can be incompatible, although the external behavior of the two versions is similar. For example, the internal representation of a container may be an array in the old version but a linked list in the new version. As a result, transformer programming not only requires a thorough understanding of implementation details in both
versions, but also has to break the principle of information hiding and manipulate low-level data representations.

In this paper we aim at automating the state transformation for DSU. In theory, it is not possible to automatically generate correct transformers for dynamic updates of arbitrary programs [17]. Nevertheless, in many practical cases, for particular software patches and particular dynamic update points, state transformations can be automatically derived with sophisticated program analysis under some proper assumptions. This kind of techniques can help reduce service disruption caused by software updates, and are useful for application domains where high availability is the major concern and occasional errors are tolerable or compensable.

Our approach, named AOTES, is designed for DSU of object-oriented programs, or more specifically, Java programs. In object-oriented programming, an important principle is to use **information hiding** and **encapsulation**. An object should be interacted only via its methods, where methods are closely related to the behavior of the object. Based on this principle, we have the following observations. First, the current state of an object is a conclusion of its past method invocations. Second, the current state is also the basis of the future method invocations. Third, the behavior of an object, or specifically the history of method invocations, is mostly unchanged during updating, especially when the patch does not include new behaviors. Thereby, the new state of a stale object (i.e., an instance of an updated class) can be synthesized by replaying its past method invocations with the new version of methods. In this way we can avoid direct mapping of concrete states between two program versions with different implementations.

Specifically, AOTES abstracts the runtime state of a stale object as a **history of method invocations** (i.e., invocation history for short) that can produce the current state from an **initial (object) state**. For example, suppose that an array based container object with three elements $e_1$, $e_2$ and $e_3$ is created by a history of $\text{add}(e_1)$, $\text{add}(e_2)$, $\text{remove}(e_2)$, $\text{add}(e_3)$ and $\text{add}(1,e_2)$. Now a dynamic update requires transforming the array based container into a linked list based one. We can easily know that the new linked list based container can be naturally acquired by applying the invocation history with the new version of methods on an empty linked list based container.

The main challenge of this approach is to obtain the invocation history for a stale object. Apparently, recording every method invocation on every potentially updated object is prohibitively expensive. Moreover, the actual history may contain redundant elements, e.g., $\text{remove}(e_2)$ and $\text{add}(1,e_2)$. To address these problems, we try to synthesize an equivalent but more compact invocation history from the current state. For the previous example, we can synthesize a history of $\text{add}(e_1)$, $\text{add}(e_2)$ and $\text{add}(e_3)$ instead of the actual one.

Unfortunately, it is hardly feasible to synthesize an invocation history using methods of real world programs. First, synthesizing a single method invocation is difficult because a method invocation generally requires arguments, which usually lie within a large value space (e.g., $[-2^{31}, 2^{32} - 1]$ for **int** in Java). Second, searching for a valid invocation history is time-consuming because method invocations need to be ordered properly to form a valid invocation history. In the scenario of dynamic updating, an additional challenge is that the synthesis is performed online and must be completed very quickly.

AOTES addresses these challenges by combining the power of symbolic execution, program synthesis and execution synthesis. Specifically, AOTES first distills a set of promising execution paths for each method by an offline symbolic execution technique. During dynamic updating, AOTES uses the selected execution paths only to realize a backward online execution synthesis technique. To reuse techniques of forward execution synthesis, AOTES...
synthesizes an inverse method for each selected execution path. We tried out AOTES on 21 real updates of widely used open source software. AOTES correctly handled 51 of 61 different transformations, while two state-of-the-art methods handled 11 and 6 of 61, respectively.

The paper makes the following primary contributions:

- We propose a mechanism to synthesize method invocation histories that can be used to recreate objects.
- We use the object recreating mechanism to automate object transformations for DSU.
- We implement the mechanism and evaluate it with updates taken from widely used open source systems.\(^1\)

The rest of this paper is organized as follows. We first give an introduction to DSU and AOTES using an illustrative example in Section 2 and then a detailed overview of AOTES in Section 3. Next, we describe the offline analysis in Section 4 and online synthesis in Section 5. Then, we illustrate the implementation of AOTES in Section 6 and evaluate AOTES with updates from real-world software in Section 7. We summarize related work in Section 8 and conclude in Section 9.

\section{Illustrative Example}

In this section, we present an introduction to DSU and AOTES using an illustrative example.

\subsection{Dynamic Software Updating and Its Challenges}

Software is subject to changes and evolution: Bugs are fixed and new features are introduced by applying software updates. Figure 1 shows a real-world motivating example of software update, which will be discussed throughout the paper. The update is from the Apache SSHD Server. Class DefaultSshFuture provides a method \texttt{addListener} to add listeners (Figure 1a). For most cases, there is only one or two listeners but apparently the implementation should support adding more. To save memory, the old version saves the first-added listener to \texttt{firstListener} and others into \texttt{otherListeners}, which is an auto-expanding list container (\texttt{ArrayList}). The new version (Figure 1b) only uses a single field \texttt{listeners} and a raw array to handle all situations.

To allow long-running programs to receive timely updates without restart, dynamic software updating migrates the running program from the old version to a new version. Specifically, a DSU system takes over the execution of a running program, transforms the runtime state at a properly determined update point (e.g., when no updated method is active) to a new state conforming to the new version, and then continues executing with the new version \cite{23,40}.

A major challenge in DSU is state transformation at the update point. A runtime system’s state consists of code, the stacks and the heap. As new code can be easily dynamically re-loaded and stacks mostly remain unchanged at the update point, the main challenge is heap state transformation. We restrict our discussions to object-oriented programming languages (e.g., Java) and thus the heap state transformation is particularly referred to as object transformation.

Object transformation takes the current state of a stale object as input and produces the new state as output. The transformation must be consistent: The future execution must be consistent with the input.

\footnote{All source code of AOTES and tests used in the evaluation are publicly available at \url{http://anonymized-url}.}
class DefaultSshFuture {
    SshFutureListener firstListener;
    List otherListeners;
    void addListener(SshFutureListener listener) {
        if (firstListener == null) {
            firstListener = listener;
        } else {
            if (otherListeners == null) {
                otherListeners = new ArrayList(1);
            }
            otherListeners.add(listener);
        }
    }
}

(a) The old version of DefaultSshFuture

class DefaultSshFuture {
    Object listeners;
    void addListener(SshFutureListener listener) {
        if (listeners == null) {
            listeners = listener;
        } else if (listeners instanceof SshFutureListener) {
            listeners = new Object[]{listeners, listener};
        } else {
            // Check the array bound
            // Expand the array if necessary
            // Append the listener
        }
    }
}

(b) The new version of DefaultSshFuture

Figure 1 An update (rev. b98697) of class DefaultSshFuture in Apache SSHD Server.

able to continue from the transformed state and take over the ongoing business smoothly. None of existing approaches [40, 41, 30, 37] is capable of automatically conducting object transformation beyond trivial cases. Most state-of-the-art DSU systems [40, 41, 37] provide default transformations that simply copy the values of unchanged fields from a stale object to its corresponding new object, and initialize all new fields with type-specific default values, e.g., 0 for int.

TOS [30] is the only known approach to automating object transformation, which embodies the idea of learning-by-example. A transformation example consists of an old-version object, which is collected during running a test over the old version of the program, and a new-version object, which is collected during running the same test over the new version of the program at the corresponding time point [30]. After collecting sufficient examples, TOS inductively composes a function following a set of predefined rules until the composed function can realize the transformations between all examples. However, TOS relies on the high quality tests in terms of covering transformations not only in testing but also in production. Even though there are sufficient good examples, TOS may easily fail due to its...
poor predefined rules.

```java
void update(DefaultSshFuture1 o, DefaultSshFuture2 n) {
    if (o.firstListener != null) {
        if (o.otherListeners == null) {
            n.listeners = new Object[] {o.firstListener};
        } else if (o.otherListeners.size() > 0) {
            int length = o.otherListeners.size() + 1;
            n.listeners = new Object[length];
            n.listeners[0] = o.firstListener;
            for (int i = 0; i < o.otherListeners.size(); i++) {
                n.listeners[i + 1] = o.otherListener.get(i);
            }
        }
    }
}
```

Figure 2 A user-defined transformer for the example in Figure 1.

Unfortunately, both default transformation and TOS do not work for our motivating example because they only use matched fields (i.e., fields with the exactly same name and type) to transfer information from the stale state to the new state. In other words, neither of them can find the relation between unmatched fields, i.e., old fields (e.g., firstListener and otherListeners) and new fields (e.g., listeners) that have different names or types. The only solution before this paper is to ask the developer to provide an object state transformer, which is a non-trivial procedure tightly coupled with program semantics and low-level implementations. Figure 2 presents a manually prepared transformer for the update in Figure 1. Even though there may be only a single stale state at the updating point, the transformer has to handle various stale object states and produces the new object states accordingly by directly manipulating the data structure of the object.

2.2 Object Transformation Using Method Invocation History

Object transformation will be easy if the method invocation history of an object is available. A method invocation consists of a method and a sequence of arguments, which may be empty if the method requires no arguments. A method invocation usually accepts some specific input state of the receiver object and produces an output state accordingly.

A method invocation history (invocation history for short) is a sequence of method invocations. Similarly, an invocation history accepts some specific initial state, i.e., the input state for the first invocation, and produces the final state, i.e., the output state of the last invocation. During replaying an invocation history, every method invocation must produce a valid output state as the input state for the consecutive method invocation in the history.

Two invocation histories are equivalent if they can yield the same final state when applied to the same initial state. For every object, there is a unique actual invocation history, including all method invocations applied to the object in the chronological order. An invocation history is complete if it can yield the current state of an object from the empty state, e.g., the actual history. Note that nested methods are not included in an invocation history. For example, method add in Figure 4 can be included in an invocation history and nested methods such as ensureCapacityInternal and ensureCapacityInternal cannot.

We have the following two assumptions for our approach:
1. The current state of an object is a summary of its past method invocations. We can also recreate the current state of an object from its past method invocations, i.e., replaying every method invocation on an object from the initial state.

2. The “role” or the behavior of an object is not changed during update [30]. The method invocation history usually keeps unchanged for such objects during updates that do not introduce new functionality, e.g., bug fix or . Hence, the new state can be easily derived from the old invocation history.

![Figure 3](object-state-evolution.png)

**Figure 3** Object state evolution of DefaultSshFuture. \( s^v_i \) denotes the \( i \)-th state of the object in version \( v \). Each state is depicted with a graphical representation of its data structure.

Now, the idea can be explained in Figure 3 by an update of DefaultSshFuture. The program invokes \( \text{addListener}^1 \) (In this paper superscripts denote program versions) with \( l_1, l_2 \) and \( l_3 \), respectively, and the update point \( (s^1_3) \) is reached. The transformed new-version object is synthesized by applying the invocation history, i.e., invoking the new-version \( \text{addListener}^2 \) with \( l_1, l_2 \) and \( l_3 \) on a newly allocated new-version object \( (s^2_0) \). State \( s^2_3 \) contains exactly the same sequence of listeners as \( s^1_3 \), indicating that this is a semantically correct state transformation. In contrast to default transformation and TOS, which cannot connect unmatched fields, AOTES finds the relations between them by matching arguments of matched methods.

One limitation of our approach is that methods in the invocation history should be available in both versions. In practice, software updates generally preserve binary compatibility. There is always a method with the same name and signature in the new version of an object whose interface is not changed. These methods are named matched methods for later discussion. Apparently, an updated class may be binary incompatible. AOTES does not insist that every changed class should preserve the binary compatibility. Suppose that a stale object’s interface is changed. The invocation of an unmatched method on the object must be eventually enclosed in an invocation of a matched method. We can enlarge the scope of the state being transformed to include all objects subject to the invocation of the matched method and use the matched method for history synthesis. In practice, the scope for updating is quite small, including one or two classes, since dynamic updating is often used for evolutionary changes (e.g., build-to-build) rather than revolutionary changes (e.g., release-to-release), and build-to-build changes usually do not introduce large patches.

### 2.3 Synthesizing the Equivalent Invocation History

Recording method invocation history for state transformation is impractical for long-running programs, because (1) we could not predict which objects were to be updated, thereby all method invocations on all objects would have to be recorded, which would introduce
significant runtime overhead; (2) the log would be prohibitively expensive to store; and (3) replaying a long history could lead to a large service disruption during updating.

Alternatively, we try to find an *equivalent invocation history* that yields the same object state as the actual invocation history when applied to an object in the initial state, but is more *compact*, in terms of as few as possible redundant method invocations. For example, listeners can be added and removed for millions of times for a `DefaultSshFuture` object. However, at a specific execution point, only limited listeners are expected in the data structure. Any invocation history that yields exactly the same set of listeners suffices for a consistent object transformation.

**AOTES** synthesizes an object’s equivalent invocation history from its current state without any logging. No history data need to be kept at runtime and no overhead is introduced when the program is not being updated. However, the synthesis of an invocation history is non-trivial because we need to first derive the arguments for a single method invocation and then find a valid history of method invocations. That means each method invocation must produce a valid output state as the input state for its consecutive method invocation in the synthesized history and the final state must be the current state of the object.

A naïve approach would enumerate all possible combinations of methods and arguments to determine the set of all possible method invocations. Since this searching space of invocation histories is huge, it is expensive to find an invocation history that can realize the state transition from the empty state to the current state of a given object. To narrow down the searching space for arguments [43, 5], execution synthesis techniques leverage on symbolic execution and constraint solver. However, these approaches aim at searching for an execution path that reaches a particular statement, while **AOTES** aims at searching for an execution path that produces the given output state on a given input state of the receiver. In addition, these approaches are used for offline scenarios such as crash reproduction and search in the space of all execution paths, which may lead to a potentially unbounded searching time for real-world programs.

We observed that multiple execution paths of a method actually have the same purpose. To derive arguments of a method invocation, **AOTES** applies an offline analysis to populate a set of promising execution paths before dynamic updating, and during dynamic updating applies an online execution synthesis that considers these execution paths only. Typically, a method with multiple paths usually has a *fast path* that handles the most common situation and many *slow paths* that handle the rest cases. Moreover, the length of the execution path is usually guided by some input. We found that first the fast path is sufficient during execution synthesis for some methods, and second a long execution path guided by a large input can be replaced by many short execution paths guided by a small input.

Take the program in Figure 4 as an example. Method `add` has a fast path that appends the added element directly into the array (at line 6) and many slow paths that need to additionally calculate the new array size and expand the array (at line 19). **AOTES** can use the fast path only to synthesize the invocation history as if the array is initially allocated in the current size without any expansion. By this way, we can exclude the slow path (*i.e.*, the call to `grow`) during execution synthesis. The fast path of `add` can be expressed by the method in Figure 5. Besides, an invocation of `addAll` with a large input collection in the actual history can be replaced by many invocations of `addAll` with a small input collection, or even many invocations of `add` with a single element.

To avoid backtracking, **AOTES** conducts a greedy backward searching starting from the current state instead of a forward searching starting from the initial state. Instead of every step searching for a method invocation that is applicable to a given input state, **AOTES**
class ArrayList<E> {
    int modCount, size, minCapacity;
    E[] elementData

    public boolean add(E e) {
        ensureCapacityInternal(size + 1); // Increments modCount!!
        elementData[size++] = e;
        return true;
    }

    private void ensureCapacityInternal(int minCapacity) {
        if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
            minCapacity = Math.max(DEFAULT_CAPACITY, minCapacity);
        }

        ensureExplicitCapacity(minCapacity);
    }

    private void ensureExplicitCapacity(int minCapacity) {
        modCount ++;
        // overflow-conscious code
        if (minCapacity - elementData.length > 0)
            grow(minCapacity);
    }

    private void grow(int capacity) {...}

    public boolean addAll(Collection<? extends E> c) {...}
}

Figure 4 Simplified version of class ArrayList in JDK.

public boolean add(E e) {
    modCount ++;
    if (!(minCapacity - elementData.length > 0))
        elementData[size++] = e;

    return true;
}

Figure 5 The simplified equivalent version of method add.

searches for a method invocation that can produce a given output state. This is because the initial input state (i.e., the empty state) has zero information to guide the search, while the final output state (i.e., the current state) has fruitful information. For example, if we synthesize a history for an array list from the empty state, we may include many method invocations that add or remove irrelevant elements. But if we synthesize the history backward from the current state, we can require that every method invocation must at least contribute to a field with a non-default value in the current output state.

To facilitate the backward searching, AOTES converts each execution path into a separated inverse method by the offline analysis. The benefit is that we can simply make use of existing symbolic execution techniques to realize backward execution synthesis. The output state of the original method is used as the input state for the inverse method. The input state of the original method is the output state of the inverse method. Figure 6 shows an inverse method `addAll` of the method `add` shown in Figure 5. Here, an inverse method takes no arguments, reverts the receiver to a previous state, and finally returns a sequence of arguments that can be used to replay the invocation history of original methods.
```java
Object[] add() {
    size--;  
    modCount--;  
    assert (!(minCapacity - elementData.length > 0));
    return new Object[] { elementData[size] };
}
```

**Figure 6** The inverse method of `add` shown in Figure 5.

**Figure 7** System overview.

### 3 Approach Overview

Figure 7 presents an overview of our approach. AOTES aims at automatically constructing the new state $s_n$ based on the current state $s_c$ only. There must be an actual history $H^1_a$ that leads to $s_c$. Instead of recording $H^1_a$ from scratch, AOTES tries to synthesize a history $H^1_s$ that can also lead to $s_c$. As the state of an object is a summary of its past method invocations, $H^1_a$ and $H^1_s$ should encode the same behavior accordingly. We assume that the role and the behavior of an object is unchanged during updating. Therefore, $H^1_s$ can be used to recreate the new state $s_n$.

AOTES first conducts an offline analysis, which takes two versions of a program ($P^1$ and $P^2$) as input, and tries to produce the following output:

- A number of execution summaries $(e_m)$ for each matched method $m$ in $P^1$. Using symbolic execution, AOTES builds a map $M_m : S \times P \rightarrow S$ from the symbolic pre-state $\Sigma_{pre} \in S$, i.e., symbolic object state before applying this method, and the symbolic arguments $\Psi \in P$, to the symbolic post-state $\Sigma_{post} \in S$, i.e., symbolic object state after applying this method.

- A number of inverse execution summaries $e^{-1}_m \in \overline{M}_m : S \rightarrow S \times P$, each of which corresponds to an execution summary in $M_m$. An inverse execution summary takes a concrete state $s$ aligned with the symbolic post-state $\Sigma_{post}$ and computes a concrete state $s'$ aligned with the symbolic pre-state $\Sigma_{pre}$ and concrete arguments $p$ such that applying $m$ with $p$ on an object in state $s'$ will change its state to $s$.

To facilitate the online searching, AOTES serializes an inverse execution summary into an inverse method. An inverse method $\overline{m}$ takes only the receiver as input and reverts the state of the receiver to the state just before invoking the original method $m$. Moreover, it also returns all arguments required by the invocation of the original method $m$. For example, suppose that `addListener1` is an inverse method of `addListener1`. We can obtain...
Figure 8 Inverse methods of `addListener` in Figure 1a. The execution traces of lines 5 and 6, lines 5, 8, 9 and 11, and lines 5, 8 and 11 in Figure 1a are postfixed with 1, 2 and 3, respectively.

$\text{s}_1$ if we apply `addListener` with $l_2$ to $\text{s}_1$. Conversely, we can obtain $\text{s}_1$ and $l_2$ if we apply `addListener` on $\text{s}_2$.

Next, AOTES attempts to synthesize an inverse method invocation history (inverse invocation history for short) for the object and revert the object to the initial state. An inverse invocation history is a sequence of inverse methods and return values (i.e., the corresponding reverted arguments). For example, $(\text{addListener}_1/l_2, \text{addListener}_1/l_1)$ is a synthesized inverse invocation history for $\text{s}_2$. The inverse invocation history can be synthesized by concatenating inverse methods as all inverse methods only take the object as input. Specifically, the inverse invocation history $H_2^{\text{inv}}$ is synthesized by searching for a sequence of inverse execution summaries $e_{m_i}, e_{m_{i-1}}, \ldots, e_{m_1}$, such that $e_{m_1}(e_{m_2}(\cdots(e_{m_i}(\text{s}_0)))) = \text{s}_0$ and as well $e_{m_i}(e_{m_{i-1}}(\cdots(e_{m_1}((\text{s}_0))))) = \text{s}_e$.

Finally, AOTES constructs a new invocation history by inverting the inverse invocation history, substituting every inverse method with the new version of its original method, and using the return value of each inverse method as the arguments. The new state can be reified by applying the new invocation history on the new initial state. For example, an invocation history $(\text{addListener}_2(l_1), \text{addListener}_2(l_2))$ can be constructed by reverting the inverse invocation history $(\text{addListener}_2/l_2, \text{addListener}_2/l_1)$.

If the input of the original method cannot be derived by executing the inverse method, AOTES introduces a fresh symbolic variable for the input and leverages symbolic execution techniques to derive its value during online execution synthesis. AOTES considers a set of short execution paths as the promising candidates for execution synthesis. This is because long execution paths generally produce long path constraints that may not be solved by a constraint solver. Moreover, short paths can also help to mitigate the problem of long-running loop and deep recursive methods whose execution is guided by some input [42]. A long execution path guided by a very large input is replaced by many short execution paths, each of which is guided by a small input. An example about loop and recursion with detailed explanation is available in Section 4.4.
Figure 8 shows three inverse methods of `addListener` (in Figure 1a) generated by AOTES. Note that the generated code has been simplified for presentation. When applying `addListener` to `s`, we can obtain `s_0` and `l_1`. Specifically, `addListener` first loads `l_1` from `firstListener` at line 2 and returns it at line 5, and then updates `firstListener` with a fresh symbolic value (denoted by a wild-card `*`) at line 3. The assertion at line 4 restricts the fresh symbolic value to be `null`, which can be derived by a constraint solver. Thereby, `firstListener` is `null` and the state becomes `s_0` if the previous state is `s_1`. Every such inverse method loads values from the current state to build return values (at lines 2, 14 and 31), updates fields to revert the state (at lines 3, 11, 15, 18, 30 and 32), checks constraints using assertions, and finally returns an array of values that can be used as arguments to invoke the original method.

AOTES can sacrifice the completeness because it does not aim at synthesizing all possible transformations. The object that AOTES cannot handle may be disposed at a later update point. On the other hand, a fixed number of inverse methods are sufficient for all transformations in practice. In Section 5.1, we will show that three inverse methods are sufficient for all transformations of `DefaultSshFuture`.

4 Inverse Program Synthesis

The insight of AOTES is to synthesize an inverse method from a symbolic execution trace not from all traces. We first give a high level overview of the symbolic execution technique of AOTES followed by a detailed description before illustrating the details.

4.1 Symbolic Execution of AOTES

In general, symbolic execution [25] is a technique to interpret a program with symbolic values instead of concrete values. A symbolic value is a formula over a set of symbolic inputs, and can be evaluated to a concrete value by substituting symbolic inputs with concrete values and then evaluating the formula.

AOTES populates a certain number of symbolic execution traces from a matched method to generate inverse methods. The symbolic execution technique of AOTES needs to allocate objects with explicit types, because dynamic method dispatching should know the type of each object. This requirement makes it non-trivial to symbolically execute an arbitrary method of an object, because the heap, or at least the receiver, must be instantiated in to a proper shape before execution. We name this heap pre-heap (Π_pre).

For example, suppose that a symbolic execution trace of `addListener` in Figure 1a explores lines 5, 8 and 11. Invoking `add` at line 11 should trigger a `NullPointerException` (NPE) if `otherListeners` does not reference an object. Otherwise, we have no idea about exploring which `add` method. To avoid the NPE and continue the execution, one can allocate an object in the pre-heap for `otherListeners` before the execution. However, we have no idea about the type for object allocation as there may be numerous subclasses of `List`. Apparently, the type for object allocation should be as exact as possible. Here, the type must be `ArrayList` not any other type.

During the symbolic execution, an object is either pre-allocated in the pre-heap before execution or newly allocated during execution. The type of a newly allocated object is known at its allocation site. For pre-allocated objects, AOTES maintains a shared dictionary S that maps an access path (e.g., this.otherListeners) to a set of types. A type is randomly picked out for the pre-allocated object if there are multiple types for an entry.
Table 1 Bytecode instructions.

<table>
<thead>
<tr>
<th>Type</th>
<th>Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stack &amp; Local</td>
<td>ldc c, load i, store i</td>
</tr>
<tr>
<td>Array Access</td>
<td>aload, astore</td>
</tr>
<tr>
<td>Field Access</td>
<td>getfield f, putfield f</td>
</tr>
<tr>
<td>Allocation</td>
<td>new C, newarray</td>
</tr>
<tr>
<td>Binary Operator</td>
<td>add, sub, mul, div, rem</td>
</tr>
<tr>
<td>Branch</td>
<td>ifgt ρ, ifeq ρ, iflt ρ, goto ρ</td>
</tr>
<tr>
<td>Invoke &amp; Return</td>
<td>invoke m, return</td>
</tr>
</tbody>
</table>

The dictionary $S$ is empty at first and updated by traversing the heap at the end of every successful symbolic execution, which is named post-heap ($\Pi_{post}$). Note that at any time, only live objects in a heap are of interest. The execution trace that explores lines 5, 8 and 11 depends on the type at this.$\text{otherListeners}$ in $S$. Hence, the execution should be first suspended at line 11 and resumed until some other symbolic execution trace updates the entry. Fortunately, the execution that explores lines 5, 8, 9 and 11 can update the entry. Line 9 allocates an ArrayList for otherListeners. The entry at this.$\text{otherListeners}$ in $S$ is updated by ArrayList.

To update missing entries in $S$, AOTES dynamically collects extra methods to execute. If the missing entry is rooted at the receiver, all methods of the receiver are added. If the missing entry is rooted at an argument of the entry method of the symbolic execution, all callers of the entry method are added. Callers of a method are determined by a call graph. AOTES constructs a static call graph at first and refines it when invoking a method during symbolic execution.

4.2 Program and Execution Definitions

This subsection gives a detailed description of the symbolic execution technique of AOTES. A program in AOTES is a set of classes. A class $C$ is a set of fields $F$ and a set of methods $M$, which also include those inherited from super classes. Every method has a receiver and an optional sequence of parameters. A method is a sequence of Java virtual machine bytecode instructions [29]. A bytecode instruction may allocate new objects, create new values, copy or move existing values, and evaluate branch conditions and change control flow accordingly.

We group all bytecode instructions into seven groups, which are shown in Table 1. A bytecode instruction may have one operand encoded with it. In a nutshell, this kind of operand may be an array index $i$, a field $f$, a method $m$, a class $C$, a constant $c$, or an offset $\rho$ of instruction index. AOTES can handle almost all bytecode instructions, except invokedynamic. This is because invokedynamic usually needs to execute a piece of custom code to resolve the callee. We list a single invoke instruction only without showing various method dispatching semantics (e.g., invokevirtual and invokesspecial), because AOTES tracks the type of every object and method dispatching for these invoke instructions is straightforward if we know the receiver type.

An object, i.e., an instance of a class or an array, is defined as a tuple of $(C, L)$, in which $C$ is its type and $L$ is its heap locations. $C$ is either the class of the instance, or a generic array type, which means that we do not distinguish array element types. A variable that can appear in symbolic input and output (actually as a fresh symbolic value) is represented by a location $θ$, which may be a named location or a heap location of an object. A named location is either the receiver or a method parameter of the entry method of the symbolic execution. A heap location is either an object field or array element and denoted by $(o, α)$,

[License Information]
in which \( o \) is the reference to its object, \( \alpha \) is either a field \( f \) or a symbolic value \( i \) representing the array index.

AOTES uses a value node to represent a symbolic value. A value node is made of a tuple of \((t, P, a)\), in which \( t \) is the type of the node, \( P \) is a set of predecessors, \( a \) is an optional type-specific attribute associated with this node. There are six types of value nodes.

1. **Constant**: \((\text{CONST}, \emptyset, c)\), where \( c \) is the constant literal in an \texttt{ldc} instruction.
2. **Reference**: \((\text{REF}, \emptyset)\). The heap is a mapping between values and objects. A new or \texttt{newarray} instruction allocates a new object and creates a reference value for the object to retrieve the object from the heap.
3. **Expression**: \((\text{EXPR}, \{v_1, v_2\}, op)\), where \( v_1 \) and \( v_2 \) are two operand values and \( op \) is a binary operator, \( i.e. \), one of \(+, -, \ast, /, \%\), and \( \# \).
4. **Assertion**: \((\text{ASSERT}, \{v_1, v_2\}, op)\), where \( v_1 \) and \( v_2 \) are two operand values and \( op \) is a relational operator, \( i.e. \), one of \( \rangle, \rangle=, \rangle=, !=, \langle \) and \( \langle= \). An opposite operator (\( e.g. \), \( \langle= \)) is used when a false branch is taken.
5. **Input**: \((\text{INPUT}, \emptyset, \theta)\), where \( \theta \) is a location in the pre-heap or a method parameter.
6. **Output**: \((\text{OUTPUT}, \{v_0\}, \theta)\), where \( \theta \) is a location in the post-heap, and \( v_0 \) is the value of \( \theta \).

Figure 9 summarizes the effects of each bytecode instructions in terms of the modification of a configuration. A configuration is a reflection of the runtime of a running Java program, and is composed of the following components, denoted as \((\mathcal{F}, \Pi, \Phi, \Sigma)\):

- \( \mathcal{F} \), the stack for method frames.
- \( \Pi \), the symbolic heap, a mapping from values to objects.
- \( \Phi \), the path condition, actually a sequence of \text{ASSERT}.
- \( \Sigma \), the symbolic state, a mapping from variable (locations) to values (nodes).

A method frame is denoted by a tuple \((m, pc, \mathcal{L}, \mathcal{E})\) of the method \( m \), the current bytecode index \( pc \), the local variables array \( \mathcal{L} \), and the expression stack \( \mathcal{E} \) for bytecode instructions [29]. Since most instructions are intra-procedure, we ignore the method \( m \) and a configuration is also denoted by a sextuple \((pc, \mathcal{L}, \Pi, \Phi, \Sigma)\).

Rules in Fig. 9 actually define an structural operational semantics [34] of each bytecode instruction over the value node. The detailed semantics of each bytecode instruction can be found in [29]. Table 2 summarizes the symbols used in describing every rule.

### Table 2 Symbols used in the rules

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \sigma, \sigma' )</td>
<td>a configuration ((\mathcal{F}, \Pi, \Phi, \Sigma))</td>
</tr>
<tr>
<td>\texttt{ldc} ( c, \sigma ) \Rightarrow \sigma'</td>
<td>a rule for the instruction \texttt{ldc} ( c )</td>
</tr>
<tr>
<td>( v, o, i )</td>
<td>a generic value, a reference value and an index value, respectively</td>
</tr>
<tr>
<td>( \Pi[0] )</td>
<td>obtain the object referenced by ( o )</td>
</tr>
<tr>
<td>( \Pi[o=(C, L)] )</td>
<td>update the heap and make ( o ) reference the object ((C, L))</td>
</tr>
<tr>
<td>( \Sigma[o, i] )</td>
<td>read the value of the location ((o, i))</td>
</tr>
<tr>
<td>( \Sigma[(o, i)=v] )</td>
<td>update the value of the location ((o, i))</td>
</tr>
<tr>
<td>( \mathcal{F} \cdot (m, pc, \mathcal{L}, \mathcal{E}) )</td>
<td>push a frame ((m, pc, \mathcal{L}, \mathcal{E})) to the method frame stack ( \mathcal{F} )</td>
</tr>
<tr>
<td>( \mathcal{E} \cdot v )</td>
<td>push a value ( v ) into the expression stack</td>
</tr>
</tbody>
</table>
Stack & Locals

\( \text{idc} \ v, (pc, L, E, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot (\text{CONST}, \emptyset, \emptyset), \Pi, \Phi, \Sigma) \)

\( \text{load} \ i, (pc, L, E, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot [i], \Pi, \Phi, \Sigma) \)

\( \text{store} \ i, (pc, L, E \cdot v, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L[i = v], E, \Pi, \Phi, \Sigma) \)

Array Access

\( \text{aload} \ (pc, L, E \cdot o \cdot i, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot \Sigma[(o, i)], \Pi, \Phi, \Sigma) \)

\( \text{astore} \ (pc, L, E \cdot o \cdot i \cdot v, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E, \Pi, \Phi, \Sigma[(o, i) = v]) \)

Field Access

\( \text{getfield} \ f, (pc, L, E \cdot o, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot \Sigma[(o, f)], \Pi, \Phi, \Sigma) \)

\( \text{putfield} \ f, (pc, L, E \cdot o \cdot v, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E, \Pi, \Phi, \Sigma[(o, i) = v]) \)

Allocation

\( \text{new} \ C, (pc, L, E, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot o, \Pi[o = (C, L)], \Phi, \Sigma) \land o \leftarrow (\text{REF}, \emptyset) \)

\( \text{newarray} \ (pc, L, E \cdot v, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot o, \Pi[o = (L, k)], \Phi, \Sigma[(o, i) = v]) \land o \leftarrow (\text{REF}, \emptyset) \)

Binary Operator

\( \text{add} \ (pc, L, E \cdot v_1 \cdot v_2, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot (\text{EXPR}, \{v_1, v_2\}, +), \Pi, \Phi, \Sigma) \)

Branch

\( \text{ifgt} \ p, (pc, L, E \cdot v_1 \cdot v_2, \Pi, \Phi, \Sigma) \rightarrow (pc + p, L, E \cdot (\Phi \cup \{\text{ASSERT, } \{v_1, v_2\}, \lambda\}, \Sigma), \text{if true}) \)

\( \text{ifgt} \ p, (pc, L, E \cdot v_1 \cdot v_2, \Pi, \Phi, \Sigma) \rightarrow (pc + 1, L, E \cdot (\Phi \cup \{\text{ASSERT, } \{v_1, v_2\}, \text{eq}\}, \Sigma), \text{if false}) \)

\( \text{goto} \ p, (pc, L, E \cdot \Pi, \Phi, \Sigma) \rightarrow (pc + p, L, E \cdot \Pi, \Phi, \Sigma) \)

Invoke & Return

\( \text{invoke} \ m, (F \cdot (m', \text{pc}, L', E' \cdot a \cdot v_1 \cdot v_m), \Pi, \Phi, \Sigma) \rightarrow (F \cdot (m, 0, L[0 = a][1 = v_1] \cdots [n = v_n], \emptyset), \Pi, \Phi, \Sigma) \)

\( \text{return} \ (F \cdot (m', \text{pc}', L', E' \cdot a \cdot v_1 \cdot v_m) \cdot (m, \text{pc}, L, E \cdot v), \Pi, \Phi, \Sigma) \rightarrow (F' \cdot (m', \text{pc}', L', E' \cdot v), \Pi, \Phi, \Sigma) \)

Figure 9 Rules describing effects of bytecode instructions. Each rule is in the format \( \langle \text{inst}, \sigma \rangle \Rightarrow \sigma' \), where \( \text{inst} \) is a bytecode instruction, \( \sigma \) and \( \sigma' \) are the configurations before and after execution the instruction, respectively.

Not all bytecode instructions create value nodes, e.g., an invoke only copies arguments from the caller to the callee. For the entry method, AOTES creates a CONST and allocates a pre-allocated object for its receiver, and creates an INPUT for each method parameter of it. INPUT in pre-allocated objects are created when first used. At the end of a normally terminated execution, AOTES creates an OUTPUT for every heap location in objects reachable from the receiver. Exceptional executions are abandoned.

Note that in symbolic execution, which branch (i.e., true or false) is taken is not determined by evaluating the condition to a concrete value but by a strategy. AOTES takes a random strategy to explore branches and collect path conditions. First, it randomly takes an unvisited branch. After all branches have been visited, it then randomly takes a visited branch. For any method, we only collect a path condition of limited length. Loop and recursion are discussed in detail in Section 4.4.

Finally, we create a value graph to summarize an execution. A value graph is a directed multi-graph in which nodes are value nodes. There are two kinds of edges between nodes, representing value dependency or location dependency. The location dependency tracks values in heap locations of INPUT and OUTPUT, e.g., object reference and array index, and are used to align the symbolic post-heap to a concrete heap. To construct the value graph, all INPUT, OUTPUT, ASSERT are first added to the value graph. Other nodes are recursively added by following the two kinds of dependency edges.

Figure 10 depicts three value graphs of addListener in Figure 1a. For better presentation, we simplify the implementation of method add of class ArrayList by removing the auto expanding branches but AOTES can handle the actual one. Let's take the left-most
Figure 10 Value graphs of three execution traces of `addListener` in Figure 1a. Their inverse methods are in Figure 8.

AOTES aims at deriving values for the two `Input` nodes from a given concrete object. The derivation is conducted by traversing the graph. First, AOTES derives a concrete value for `Output`, which can be easily done by aligning the concrete object to `this` and then loading the field `firstListener`. Next, we can derive the value of the parameter (`Input`) using the value of `firstListener` (`Output`) directly because the latter value is assigned from the former value. Note that we need to check whether the `Assert` satisfies. The `Input` (w.r.t. `firstListener`) has been overridden during forward execution by the `Output` (w.r.t. `firstListener`). We then create a fresh symbolic value for the `Input` and try to use the constraint solver to derive a value for it. All deriving steps are serialized into a method to facilitate the online execution synthesis, which will be discussed in the next subsection.

4.3 Inverse Method Synthesis

We say that a value node is `resolved` if its concrete value has been derived. An inverse method is created by resolving all `Input`. An `Output` can be directly resolved if its symbolic location can be `aligned` to a concrete location. The receiver can be directly aligned. An object field can be aligned if its object reference is resolved. Thus, all fields of the receiver can be directly aligned. An array element can be aligned if both its object reference and index are resolved. AOTES can translate every resolution and alignment into a statement. All statements finally make up the inverse method. AOTES supports four kinds of resolution methods.

1. Direct Resolution: All `Const` and aligned `Output` can be directly resolved.
2. Forward Resolution: A node is resolved if all predecessors are resolved. For example, if \(a = b \times c\), and \(b\) and \(c\) are resolved, then \(a\) can also be resolved by evaluating \(b \times c\) again.
3. Backward Resolution: If an `Expr` and one of its predecessors are resolved, we can resolve the other predecessor by these two nodes. For example, if \(a = b - c\), and \(a\) and \(b\) are
Algorithm 1: Resolution of a value graph.

**Input:** $(V,E)$, the value graph.

**Output:** $(R,U,\overline{m})$, where $R$ is the set of resolved nodes, $U$ is the set of unresolved nodes, and $\overline{m}$ is the inverse method.

1. $(R,U,\overline{m}) \leftarrow (\emptyset,V,\emptyset)$
2. repeat
3.   repeat
4.     foreach $v \in U$ do
5.       FORWARD($v,(R,U,\overline{m})$)
6.     until $R$ is fixed
7.   foreach $v \in U$ do
8.     if AGGRESSIVE($v,(R,U,\overline{m})$) then break
9. until $R$ is fixed
10. return $(R,U,\overline{m})$

resolved, then $c$ can be resolved by evaluating $b - a$. We treat $\ast, -, \ast,$ and $/ \text{ invertible}$ due to the aggressive nature of AOTES.

4. **Aggressive Resolution:** As an inverse method is used for execution synthesis, we can aggressively guess a value for an `INPUT` by assigning a fresh symbolic value to it. Besides, we can guess an index for an array element if its object reference has been resolved.

Algorithm 1 aims at resolving all nodes of a value graph. The algorithm maintains a sequence of statements $\overline{m}$, and two sets of nodes, $R$ and $U$, i.e., the sets of resolved and unresolved nodes, respectively. At first, $R$ and $\overline{m}$ are empty, and $U$ contains all nodes in the value graph. The four resolution methods try to apply rules defined in Figure 11 and return `true` if there is an applicable rule, which means some nodes have been resolved. Every successful resolution appends a statement to $\overline{m}$ (denoted by $\cup$). In theory, we can continue apply aggressive resolution to resolve every `INPUT` and then use forward resolution to resolve all unresolved nodes in the value graph. The algorithm can finally terminate when $R$ is fixed, since a value can never be moved from $R$ to $U$. $\overline{m}$ is successfully generated only if $U$ is empty. We then decorate $\overline{m}$ into a valid Java method. This method has no parameter and returns all reverted arguments.

Every value node is indeed converted into a variable with a unique name. Every resolved `INPUT` must be aligned first and its concrete location is also updated with the resolved value. For presentation, this requirement is not expressed in the rules. A fresh symbolic value is denoted by $\ast$ but in fact produced by a runtime method. We also provide a runtime method `guess` that chooses an index in a given array.

Figure 11 presents rules that are used to resolve a value node. A rule takes a value node from the graph and the currently visiting status (i.e., the tuple $(R,U,\overline{m})$) as input to update the visiting status for next visiting and produce a statement for the inverse method $\overline{m}$ as output. Each rule has a precondition that should be checked first. Basically, the precondition at least ensures that each node is resolved once by a rule. Take rules of direct resolution as an example. To resolve a `Const`, the rule only checks whether the node being resolved has been resolved. To resolve an `OUTPUT`, the two rule further check whether the location has been aligned.

Figure 8 shows the inverse methods created by resolving nodes from value graphs in Figure 10. We have simplified the output, e.g., remove redundant variables for `Const`. Algorithm 1 and rules in Figure 11 ensure that a value node is only resolved once. In fact, a value node can be resolved via different rules and nodes. For consistency, AOTES attempts to resolve every node using a different method at last and adds assertions to ensure that all resolved concrete values must be equal, e.g., lines 16 and 17.
Rules for resolving node and generating statements. Each rule is in the format
\[ (v, \theta, R, U, \overline{m}) \Rightarrow (v', \theta', R', U', \overline{m}') \],
where \( P \) is the pre-condition of applying the rule, \( v \) is the node that we attempt to resolve its value, \( R \) is the set of resolved nodes, \( U \) is the set of unresolved nodes, \( \overline{m} \) is the sequence of generated statements, and \( P', U' \) and \( \overline{m}' \) are new versions after applying the rule.

4.4 Loop and Recursion

AOTES takes a single-path symbolic execution and limits the length of the path condition. Hence, the loop and recursive method invocations are unrolled for a limited length. A set of short execution paths is considered as the promising candidates for online execution synthesis. Obviously, there do exist long-running loops and deep recursions. Thus, the symbolic execution trace may be infeasible for some inputs (pre-heaps).

Actually, the problem of loop and recursion may not be as critical as it seems to be. Recall that AOTES has no need to synthesize an inverse method for all execution traces. Besides, a fixed number of inverse methods are sufficient sometimes. In comparison with existing whole program execution techniques [43, 5], the insight of invocation history synthesis is that it infers the sequence from the state only and requires no complete control flow and call graph. Moreover, many loops and recursive methods are guided by some input [42]. AOTES can split a loop with a very large input in the actual history into many loops with a small input in the synthesized history.

Take the class in Figure 12 as an example. Method \( \text{addN} \) has a loop and also recursively calls itself. AOTES can easily populate the execution trace where \( n \) is 1 and also synthesize an inverse method for it, because \( \text{addN}(a, 1) \) is equivalent to \( \text{elements.add}(a) \). We have shown AOTES can easily handle it. Hence, no matter how divergent the actual history is, AOTES can always guarantee to synthesize a history \( \text{addN}(e_0, 1), \ldots, \text{addN}(e_{i-1}, 1) \), where \( e_i \) is the \( i \)-th element in \( \text{elements} \) and \( n \) is the size of \( \text{elements} \). For example, an \( \text{addN}(\{a, b\}, 2) \) in the actual history would be replaced by the following synthesized history: \( \text{addN}(a, 1), \text{addN}(a, 1), \text{addN}(b, 1), \text{addN}(b, 1). \)
```java
class LoopAndRecursion {
    List elements = new ArrayList();
    void addN(Object o, int n) {
        if (n < 1) {
            return;
        } else if (o instanceof Object[]) {
            for (Object e : (Object[]) o) {
                addN(e, n);
            }
        } else {
            elements.add(o);
            addN(o, n-1);
        }
    }
}
```

![Figure 12 An example of loop and recursion.](image)

**Algorithm 2:** History synthesis.

Input: $o$, the receiver object for history synthesis, $\mathcal{M}$, the set of inverse methods.

Output: $H$, the invocation history for $o$.

1. $\mathcal{H} \leftarrow \emptyset$

2. while isNotEmptyState($o$) do

3.     $T \leftarrow \emptyset$

4.     foreach $m \in \mathcal{M}$ do

5.         $a \leftarrow \text{TestApply}(m, o)$

6.         $T \leftarrow T \cup \{(m, a)\}$

7.     if $T = \emptyset$ then

8.         break

9.     $(m, a) \leftarrow \text{Rank}(T)$

10. $\mathcal{H} \leftarrow \mathcal{H} \cup \{(m, \text{Apply}(m, o))\}$

11. return $\text{Revert}(\mathcal{H})$

5 Execution Synthesis

This section depicts the online synthesis and replaying of invocation histories.

5.1 Online Synthesis of Invocation Histories

AOTES uses a greedy strategy to search for an inverse invocation history. As shown in Algorithm 2, AOTES first collects all applicable inverse method invocations ($T$), and uses a heuristic method to rank them (by function $\text{Rank}$). Intuitively, a better inverse method should revert more input in the $\Pi_{\text{pre}}$ from non-default values to default values and preserve more locations in the $\Pi_{\text{post}}$, which is the $\Pi_{\text{post}}$ for the next step. Hence, AOTES prefers the inverse method with no aggressively resolved input first, then more live locations after execution, and finally more reverted input. The searching stops when the object is in the empty state or there is no applicable inverse method. AOTES executes an inverse method in two ways, i.e., $\text{TestApply}$, which will restore the object state for applying next inverse method, and $\text{Apply}$, which will retain the modification.

For example, suppose that an object of $\text{DefaultSshFuture}$ is in state $s_3^1$. $\text{addListener}_2^1$ is not applicable as line 17 in Figure 8 fails, i.e., the size is not 2. Both $\text{addListener}_1^1$ and $\text{addListener}_3^1$ are applicable, but we prefer $\text{addListener}_3^1$ over $\text{addListener}_1^1$ as it reverts more locations and also preserves other listeners. The object state then becomes...
AOTES realizes object transformation as follows. Given a stale object, we first try to synthesize an inverse invocation history for it. If the history is empty, then we fall back to default transformation. Otherwise, we apply a default transformation to the object after reverting its state by applying the inverse invocation history. Finally, we invert the inverse invocation history to build a new history and apply it to the object. Note that the synthesized history is not necessarily to be complete.

6 Implementation

We implemented AOTES, including the symbolic execution engine, inverse method synthesizer and invocation history synthesizer, in about 25K lines of Java code. AOTES is fully automated and only takes binary class files as input and thus requires no source code, no test case and no human specified update points.

We implement a trivial single-variable solver. For example, a fresh symbolic value for int includes all integers in \([\text{MIN\_INT}, \text{MAX\_INT}]\). As such a symbolic value is mostly used in assertions and pre-states. Therefore, it narrows its range towards passing the assertion and to the default value during evaluation. For example, suppose that a variable v1 has a fresh symbolic value for int. After evaluating the assertion \(\text{assert}(v1 == 0)\), its range is narrowed to \([0, 0]\), which means that this symbolic value can only be 0. Currently, we are working on integrating Z3 [9] as the solver to further improve the effectiveness of AOTES.

The main limitation of AOTES in analyzing real-world applications is uninterpreted native methods. The current implementation of AOTES only handles a small part of native methods that we have encountered during evaluation, among which some are re-implemented using Java, e.g., arraycopy, and others are manually marked as operators, e.g., sin and identityHashCode. Operator methods are not interpreted during symbolic execution and their effects are recorded like an operator (i.e., creating an Expr node). During synthesis, they can be executed as all arguments are available at present. We allocate a phantom object for every class as the container for static fields. The reference to the phantom object is treated as a Const and thus a static field can be easily aligned.

7 Experiments

We evaluated AOTES’s effectiveness with real-world updates and performance in synthesizing long histories using a micro benchmark, respectively. All experiments were conducted on an Intel Core i7 3.4GHz machine with 20 GB memory running 64-bit Windows 10. The offline synthesis was conducted on JDK 1.8.0_65 and the dynamic updating was carried out on Javelus. We forced AOTES to only explore at most 20 different traces for a method and 1000 branches for a trace.
7.1 Real-world Updates

We collected 21 updated classes from Apache Commons Collections, Apache FTP Server, Apache SSHD Server and Apache Tomcat, which are all widely used common libraries and server applications under years of active development. These updates were chosen for the following reasons. First, the two versions of all updates must be successfully compiled. Second, all updates must involve field changes otherwise would require no transformation. We classified these fields changes into the following four types:

1. **Value Change**: with no field added, but the values of some fields need to be updated.
2. **Name Change**: with a field renamed only.\(^2\)
3. **Type Change**: with a type-changed field only.
4. **Complex Change**: any other changes.

Third, an updated class must not invoke uninterpreted native methods beyond those handled by the current implementation of AOTES. Finally, we also exclude rare cases in which the current state does not contain sufficient information to determine the new state. In this situation, even a programmer may not be able to provide a transformer, not to mention TOS or AOTES.

In addition to existing test cases, we additionally wrote a few test cases for some updated classes under our test frame work designed for DSU, because most updated classes have no test case and some existing test cases were insufficient to detect improper object transformation. Every test created an object with one or a few method invocations before dynamic updating. Then, we triggered the dynamic updating and applied the transformation to the object. Every dynamic update was verified as follows. That is, the state after dynamic updating of the old version must be equivalent to a state that can be achieved by executing the same methods on the new version.

We ran all tests with dynamic updating for both AOTES and default transformation on Javelus. All results are shown in Table 3. AOTES succeeded in 51 (83.6\%) updates and failed in other 10 updates due to incomplete or inconsistent synthesized histories. An incomplete history cannot update all fields as the searching also stops when no applicable inverse method. This is mainly because many native methods prevent AOTES from generating sufficient inverse methods. We plan to model more native methods in future. An inconsistent history leads to the same state as the actual history in the old version but different states in the new version. We will discuss this limitation with examples in the following paragraphs.

We did not run TOS with dynamic updating as TOS is not fully automated and requires extra training tests and manually specified update points. Instead, we used our validation tests to train TOS and hope that it could synthesize a conditional transformer for each update that can realize transformations for all test cases. TOS failed to synthesize a function for 16 of 21 updates (marked with N.A. in Table 3). For the rest 5 updates with 12 tests in total, TOS even failed in validating its output against 6 training tests.

As shown in Table 3, almost all updates have field name changes or type changes. These updates are the major of updates that require transformations in practice. Default transformation and TOS failed to derive a valid transformation/transformer even for many name and type changes because they cannot find the relations between fields with different names or types. AOTES can infer their relations when changed fields used the same arguments in matched methods. Moreover, both default transformation and TOS used a set of predefined

\(^2\) Note that if either the name or type of a field is changed, it is considered as deleted and a new field with the new name or type is added.


Table 3 Results of real-world updates.

<table>
<thead>
<tr>
<th>Type Change</th>
<th>Update</th>
<th>Tests</th>
<th>AOTES</th>
<th>Default</th>
<th>TOS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Change</td>
<td>tomcat-dd741c</td>
<td>6</td>
<td>4^a</td>
<td>4</td>
<td>N.A.</td>
</tr>
<tr>
<td>Name Change</td>
<td>ftp-5d5592</td>
<td>4</td>
<td>3^a</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>sshd-6f8507</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>tomcat-9510e8</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>tomcat-b75f5c</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>N.A.</td>
</tr>
<tr>
<td>Type Change</td>
<td>collections-0e1140</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>collections-cdacd4</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>ftp-8110b</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>sshd-054334</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>tomcat-480edc</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td>Complex Change</td>
<td>collections-b7327a</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>ftp-43f5f</td>
<td>4</td>
<td>2^a</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>ftp-4907aa</td>
<td>4</td>
<td>2^a</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>ftp-5df186</td>
<td>4</td>
<td>2^a</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>sshd-009d83</td>
<td>5</td>
<td>5</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>sshd-1487b0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>sshd-2297b2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>sshd-b98694</td>
<td>7</td>
<td>7</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>sshd-eeedee4</td>
<td>1</td>
<td>0^a</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>tomcat-24becd</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>tomcat-2db0f7</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>N.A.</td>
</tr>
<tr>
<td></td>
<td>_Total</td>
<td>61</td>
<td>51/83.6%</td>
<td>11/18.3%</td>
<td>6/9.8%</td>
</tr>
</tbody>
</table>

^a) α and β indicates inconsistent and incomplete synthesized history, respectively.

simple rules, and cannot handle transformations involving custom type conversions (e.g., ArrayList to ConcurrentHashMap). AOTES leveraged program code to infer custom type conversions when objects of different types used the same arguments in matched methods for initialization.

We discuss the effectiveness and limitation of AOTES with individual updates for every type in the following paragraphs.

Value Change: Tomcat dd741c

```
- private String jmxNameBase = "pool";
+ private String jmxNameBase = null;
```

This update only changes the initial value of jmxNameBase in the constructor. If the setter of jmxNameBase is not invoked, the transformation should update the value to null. AOTES failed in two test cases due to inconsistent histories. That is, both the constructor and the setter method can assign "pool" to jmxNameBase in the old version but null and "pool" in the new version. In fact, even a programmer cannot write a general transformer here because using the current state only cannot distinguish different actual histories that lead to the same current state.

Name Change: FTP 5d5592

```
- private int maxIdleTimeMillis = 10000;
+ private int idleTime = 300;

 public void setIdleTime(int idleTime) {
-    maxIdleTimeMillis = idleTime * 1000;
+    this.idleTime = idleTime;
} 
```

licensed under Creative Commons License CC-BY
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany
Except this update, we can just copy the value from a new field to the old field for all Name Change updates. AOTES failed in the only test for the same reason as Tomcat dd741c. maxIdleTimeMillis was set to 10000 by the constructor in the old version but in the new version idelTime should be 300.

**Type Change:** FTP f8110b

```java
class DefaultFtpletContainer {
  @class FpletEntry { String name; Fplet ftplet; }
+ private Map ftplets = new ConcurrentHashMap();
  public void addFtplet(String name, Fplet ftplet) {
-    ftplets.add(new FpletEntry(name, ftplet));
+    ftplets.put(name, ftplet);
  }
}
```

Type conversions between built-in types are easy, *e.g.*, `int` to `long`. However, for this update, we need the key to convert an `ArrayList` to a `ConcurrentHashMap`. AOTES can synthesize inverse methods for `addFtplet` and the constructor and also a history using them. The key can be inferred from the parameter `name` of the new version of `addFtplet`.

**Complex Change:** SSHD 009d83

```java
class AgentImpl {
  private List keys = new ArrayList();
  + private AtomicBoolean open = new AtomicBoolean(true);
  public void close() throws IOException {
-    closed = true;
-    keys.clear();
-    if (open.getAndSet(false)) {
-      keys.clear();
-    }
-  }
+    }
  public void addIdentity(KeyPair key, String comment) {
-    keys.add(new Pair(key, comment));
+    keys.add(new Pair(key, comment));
  }
}
```

This update changes both the type and name of a field. Method `close` removes all elements in `keys`. Hence, the synthesized history only needs to be a constructor and a `close` if the last method in the actual history is a `close`. For example, suppose that the actual history includes a constructor of `AgentImpl`, an `addIdentity`, and a `close`. However, a critical field `modCount` in `ArrayList`, which is used to avoid concurrent modification during iterating the list, prevents AOTES from applying the inverse constructor of `AgentImpl` if its value cannot be reverted to 0. Fortunately, the inverse method of `clear` can decrement `modCount`. As a result, the synthesized history is a constructor followed by two `close`.

### 7.2 Micro Benchmark

We selected five classes of commonly used collections and designed a micro benchmark to evaluate the synthesizing time of AOTES. Theoretically, the synthesizing time only directly depends on the current object state and the number of inverse methods but not the actual history. Thereby, we first conducted experiments with all synthesized inverse methods and
then repeated the experiments with a small set of inverse methods. The results help us to reveal solutions that can optimize the synthesizing time of AOTES.

The micro benchmark created an object of each class, filled it with a number of elements (ranged from 0, 10, …, 90), and finally synthesized a history for it. We also repeated the procedure for 10 times first to warm up the JVM. Figure 13a shows the distribution of inverse methods generated by AOTES for all public methods of every class. We first run the benchmark on all inverse methods and then repeat the benchmark with only previously used inverse methods.

The synthesizing time using all inverse methods is shown in Figure 13b. Apparently, the time depends on the size of elements in a collection. AOTES spent more than 5s in the worst case for Vector. This is mainly because our implementation heavily uses reflections and exceptions. Besides, most of the inverse methods of these classes were indeed redundant.

The number of inverse method candidates has an impact to the searching time. Figure 13c shows the number of different inverse methods (not different inverse method invocations) appeared in each history. No more than four inverse methods were actually used for all histories. That means the online synthesis wasted a certain amount of time on trying out redundant inverse methods. Note that here an inverse method may be invoked for many times. The actual number of method invocations in total were mostly the same as the number of elements.

Pruning redundant inverse methods can speed up the online history synthesis. In practice, we can prune redundant inverse methods using an automatic random testing tool [35]. Figure 13d shows the synthesizing time using only previously used inverse methods. AOTES only spent 35ms in the worst case for LinkedList and only 12ms for Vector. We believe that this synthesizing time is acceptable for practical usage and can also be further reduced with a more efficient implementation of AOTES.

![Figure 13](image_url)

Figure 13 Results of micro benchmark.
7.3 Discussion

AOTES can be used as a complement to existing techniques such as default transformation, TOS, and manual approaches, especially for name changes, type changes and other complex changes. AOTES can find the relations by matching the arguments in matched methods, while both default transformations and TOS cannot find the relations between old fields and new fields that have different names or types. Different from TOS, which requires test cases and manual efforts during collecting transformation examples, AOTES is fully automated and works purely on binaries without source code and test cases. AOTES is the only approach that can leverage the program code to infer powerful transformations. Moreover, it is an on-demand dynamic approach and can avoid synthesizing transformation that are hard to be automated but may not be encountered during dynamic updating.

The time of online execution synthesis may be a threat to AOTES, particularly when there are many stale objects. AOTES tackles this challenge by adopting a lazy updating mechanism and sharing searching strategies across different transformations. Specifically, AOTES first mitigate the disruption caused by synthesis using a lazy updating technique, which is naturally supported by Javelus. Stale objects are updated individually in an on-demand way. Second, to further reducing synthesis time for a single object, AOTES can try out methods that have already been used only. The effectiveness has been demonstrated in the micro benchmark. In other words, AOTES shares the searching strategy across different transformations, while TOS and manual approaches share the transformer.

8 Related Work

We survey related work in this section, including dynamic software updating, and program and execution synthesis.

8.1 Dynamic Software Updating

In general, DSU systems can be divided in two types, intra-process state transformation and inter-process state transfer.

Intra-process State Transformation Many DSU systems can generate default transformation and support programmers specified transformation as well [40, 13, 15]. Besides, they also provide a safety guarantee, e.g. type safety, to facilitate developing transformers [23, 33, 4, 31, 40, 14]. The transformation can be taken eagerly [40, 41], or lazily [14, 15], or both [37]. Although the size of a valid transformer may not be great [2], it should be delivered with extremely timing constraints, e.g., for security patches.

Other approaches for debugging prefer no user intervention by sacrificing the flexibility or validity [11, 24]. Gupta et al. have proved that the validity of general dynamic updating is undecidable [17]. Besides, existing programming techniques can also help transformer programming, e.g., formalization and verification [20, 26, 44] and software testing [19, 22].

Inter-process State Transfer Many challenges in implementing inter-process state transfer have made it not so popular in building DSU systems [21, 12]. Not all programs can run multiple instances of multiple versions simultaneously, particularly in a production environment [21], e.g., the Linux kernel. In contrast, this approach has been extensively studied in live migration of virtual machines [6].

Automated Object Transformation TOS [30] and TTST [12] require a pair of matched objects as example and infer transformations from these examples. TOS matches objects by
automatically inferred key fields but may fail as there may be no field, no key field, or no matched key field. TTST matches objects by naming and context but the name and context may be changed. AOTES requires no example as it uses matched methods, which makes it able to handle non-trivial cases that TTST and TOS cannot handle.

8.2 Program and Execution Synthesis

Program Synthesis and Execution Synthesis [43] have been extensively studied for years. Among them most related to AOTES are inverse program generation [10, 39] and data transformations [16, 18, 27, 38]. AOTES combines the program synthesis and execution synthesis. AOTES indeed makes use of reverse execution [3, 7, 1] over symbolic execution traces to generate an inverse program.

9 Conclusion

AOTES is an experimental approach to automating object transformation for dynamic software updating. It preserves the continuity of stateful behavior of objects whose classes are changed at runtime. The novelty of AOTES is to synthesize a method invocation history that can produce the current object state in the old version, and replay the history to get the desired state for the new version. Our preliminary evaluation shows that AOTES has the promising ability to handle software updates taken from real-world software systems. Although the current implementation of AOTES is for Java only, we believe that the general idea of AOTES can also apply to other object-oriented programming languages.

In the future, we plan to improve AOTES by supporting more native methods and searching strategies, and also conduct a thorough evaluation of AOTES with more real-world updates.
References


licensed under Creative Commons License CC-BY

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany


